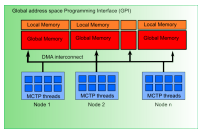

Fraunhofer ITWM

User Manual

Global Programming Interface (GPI)

Version:
1.0

Contents

1 Introduction **2**

2 Installation **2**
2.1 Requirements and platforms . . . 3
2.2 GPI Daemon . . . 3
2.3 GPI SDK . . . 4

3 Building and running GPI applications **5**
3.1 Building an application . . . 5
3.2 Running an application . . . 5

4 Programming the GPI **5**
4.1 Starting/stopping the GPI . . . 6
4.2 DMA operations . . . 7
4.3 Queues . . . 8
4.4 Passive DMA operations . . . 8
4.5 Collective operations . . . 9
4.6 Synchronisation . . . 9
4.7 Atomic operations . . . 10
4.8 Commands . . . 10
4.9 Environment checks . . . 10
4.10 Configuring GPI . . . 11
4.11 Notes on multi-threaded applications . . . 11

A Code example - envtest.cpp **12**

B Code example - transferbuffer.cpp **13**

1 Introduction

This document is intended to introduce the *Global Address Space Programming Interface* (GPI) to the application programmer and is part of the GPI SDK.

GPI provides a partitioned global address space (PGAS) to the application which in turn has direct and full access to a remote data location. The whole functionality includes communication primitives, environment runtime checks, synchronization primitives such as fast barriers or global atomic counters, all which allow the development of parallel programs for large scale.

GPI motivates and encourages an asynchronous programming model allowing for nearly perfect overlapping of communication and computation and leveraging the strengths of the different components of a computing system that is, releasing the CPU from communication whenever possible and letting the network interface asynchrounously do its task.

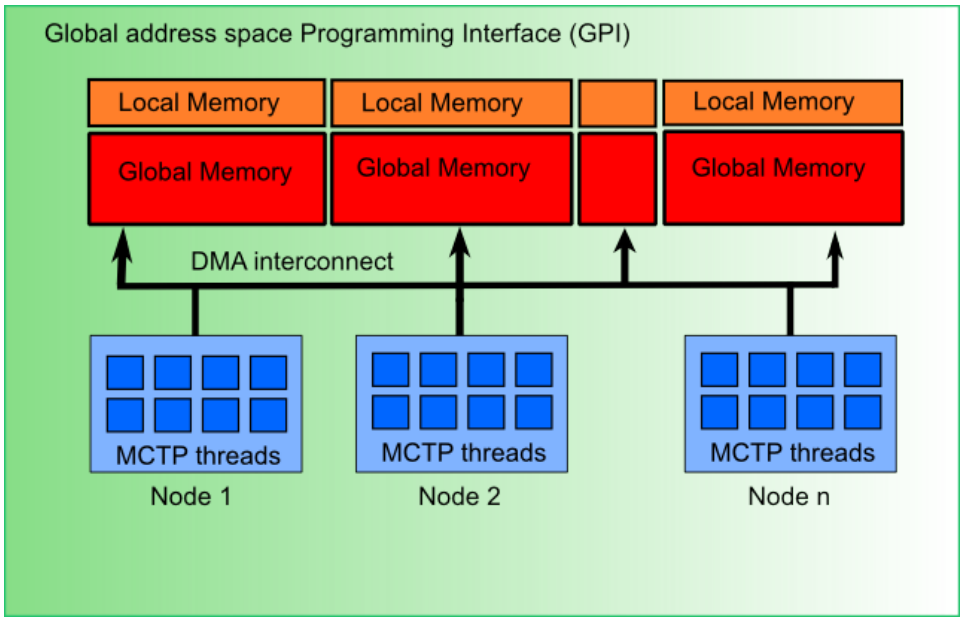

Figure 1: GPI architecture

Furthermore, the programming model of GPI promotes a threaded view of computation instead of a process-based view. As figure 1 depicts, each node is one instance of GPI running several MCTP threads - although there is no limitation to this as for example, normal pthreads might be used - where all threads have access to all partitions of the global address space and each node contributes with one partition to this global space.

2 Installation

GPI is composed of two components: the GPI daemon and the GPI SDK. The GPI daemon is an application that runs as a daemon on all nodes of a cluster and is responsible for managing GPI applications. This includes start and stop of applications, management of licenses (together with a license server) and general infrastructure control. The GPI SDK is the set of headers and libraries that an application developer needs.

Both components come with an installation and uninstallation script to simplify the installation process.

2.1 Requirements and platforms

The GPI only depends on the OFED stack from the OpenFabrics Alliance and more concretely on the libibverbs. Therefore the operating system is **Linux** and the supported Linux distributions. In terms of CPU architectures, GPI supports **x86 (64bit)**.

One requirement is that the GPI daemon runs as root (i.e. is started by root). This brings a set of advantages to the whole framework:

- full hw/driver (Infiniband, Ethernet) configuration available
- setup of Infiniband/Ethernet multicast networks
- setup of requested link-layer
- automatic resource management (pinned memory, virtual allocation, cpu sets, etc.) for GPI processes
- firmware update
- stable license management
- filecache control
- user authentication (via PAM)
- full control on GPI processes (e.g. 100% cleanup of old GPI instances, timeout, etc.). Common situations that most of batch systems have problems with
- full environment check possible (e.g. dependency check for GPI binaries)

The GPI daemon running the first node requires a *machinefile* to specify all the nodes that should be used for an GPI application. Depending on the setup of the machines, this can be user-driven or automatic. A user-driven setup refers to a small and static setup where the user controls and has privileged access to all the machines. In this case, the user edits the *machinefile* herself. On the automatic setup - the more often case - the user gets assigned a set of nodes by the batch system which in turn should also create the *machinefile* in the location where the GPI daemon is configured to search for (*/etc/gpid.conf*). This happens automatically and transparently to the user but might require a small tweak to the batch or modules system.

2.2 GPI Daemon

For the installation there is 1 script to be used: install.sh. This install.sh script installs the GPI daemon. It must be called with the **-i** option where the argument is the IP address of the license server and the **-p** option where the argument is the path where GPI is or is afterwards to be installed. It should be a directory accessible by all nodes.

The GPI daemon is distributed as a RPM that installs all the needed files on a system.

After installation, the system will have the following files:

- the gpid.exe binary installed at */usr/sbin/*
- the gpid_NetCheck.exe binary also installed at */usr/sbin/*
- the configuration file gpid.conf installed at */etc/*
- the init script gpid installed at */etc/init.d/* plus the links at runlevels 3 and 5
- the pam file gpid_users installed at */etc/pam.d/*

If the init script (*/etc/init.d/gpid*) has the correct values, then the daemon will be started after the installation.

The configuration file located at /etc/gpid.conf is required to describe the directory where the machinefile is to be found. The machinefile lists the hostnames of the machines where a GPI application is to be started.

The daemon looks for a file named machinefile. If it is not found, it will take the newest file located at the provided directory.

In a system where the user interacts with a batch system such as PBS to access computing nodes, the entry on the configuration file (/etc/gpid.conf) might look like the following:

```
NODE\_FILE\_DIR=/var/spool/torque/aux/
```

If the machinefile contains repeated entries for the hostnames, these will not be used since GPI is at the moment targeted to run one process per computing node.

The options for the daemon are configurable under /etc/init.d/gpid . The most important options are the IP address of the license server (LIC_SERVER) and the security pre-path for the binaries (PRE_PATH).

The daemon has the following starting options:

-d Run as daemon. This should always be used to start the binary as a daemon.

-p (path) Security prefix to binary folder. The security prefix describes the path to a directory to be used for starting applications.

For example: /opt/cluster/bin/

Only applications that are started in this path are allowed to run.

-a (IP address) IP address of license server (e.g. 192.168.0.254)

The license server must be running on some machine and this option describes the IP address of such machine. If this IP address is not correct and does not point to a running license server, the daemon will not be able to start.

-n (path to binary) The *gpid_NetCheck.exe* binary must be available. This binary is installed with the RPM installation and located at */usr/sbin/*. Therefore, this option is usually used as */usr/sbin/gpid_NetCheck.exe*. This binary does some infrastructure checks related to Infiniband.

-h Display the possible options for the daemon.

2.3 GPI SDK

There is 1 script to be used: install.sh. The install.sh script installs the SDK. It must be called with **-p** option where the argument is the path where to install GPI. It should be a directory accessible by all nodes.

The installation path of the GPI SDK will then have the following structure:

/include includes the header files available for application developers.

/lib64 includes the libraries for linking.

/bin is where the binaries should be placed by users in order to be able to run applications. Subdirectories herein are also allowed for a better organization of each user's binaries.

/bin/gpi_logger is the GPI logger that can be used on worker nodes to display the *stdout* output of GPI applications started by the GPI daemon.

3 Building and running GPI applications

3.1 Building an application

The GPI header *GPI.h* and the GPI library *libGPI.a* are the most important GPI components to application developers. Besides a suitable *ibverbs* library (from the OFED package) these are the only components necessary to build a GPI application. A GPI application can not start by itself, it requires the GPI daemon to run on all nodes. The GPI daemons load the binaries on the remote nodes and set up the network infrastructure. For security reasons the daemons only load binaries located in a directory with a certain *prepath* that can be specified at daemon startup. Remote nodes will subsequently be referred to by *worker nodes* while the node where a binary is started will be called *master node*.

All that is required to build a GPI application is to link the appropriate static libraries. These are *libGPI.a* and *libibverbs15.a* which are located in the 'lib64' folder where the GPI SDK was installed. Try to build the *envtest* example listed on Appendix A this document and typing (substituting and using the correct path to GPI SDK):

gcc -o envtest envtest.cpp -I<Path to GPI SDK>/include -L<Path to GPI SDK>/lib64 -lGPI -libverbs15

The next step is to run the produced binary.

3.2 Running an application

If a GPI application is executed on one node (which automatically becomes the master node) the *machinefile* of the node is checked for all participating worker nodes and each GPI daemon of the node is instructed to load and execute the binary. Note that only binaries can be run that are located in a folder with an appropriate *prepath*. Now copy the envtest binary to the appropriate directory and run it on the command line.

cp ./envtest <Path to GPI SDK>/bin/
<Path to GPI SDK>/bin/envtest

The application first executes various environment checks before starting the GPI, synchronising all nodes and shutting it down again. If an environment check fails a message describing the error is printed to *stdout*. If no errors are reported the GPI is installed correctly and you can start writing your own applications.

If there is a problem starting the binary, the following list might shed some light on the problem:

- is the GPI daemon running?
- was the binary copied to the right prepath location, where the GPI daemon can run it?
- is the daemon looking and finding the right location with the machinefile?
- is your batch system configured/modified to create the right machinefile (with the assigned nodes), at the right location for GPI?
- are you trying to run GPI on a single node? GPI is designed to run with 2 or more nodes.
- are you trying to start GPI with only a few bytes for the global memory? GPI requires at least 1 KiloByte (1024 bytes) on the gpiMemsize argument of the startGPI function.

4 Programming the GPI

The GPI interface (API) is small and with a short learning curve. The following sections summarize the API which should be consulted for complementary details.

4.1 Starting/stopping the GPI

Before any GPI operation can be executed a call to *startGPI* has to be performed. This function constructs the interconnections between all participating nodes and allocates the memory used by the GPI application (GPI memory). While a GPI application can make use of heap and stack memory like any other application, only the *GPI memory* can be the source or destination of a DMA operation. Except for this difference *GPI memory* is identical to a large continuous block of heap memory.

```
    //! Start GPI
    /*!
    \param argc Argument count
    \param argv Command line arguments
    \param cmdline The command line to be used to start the binaries
    \param gpiMemSize The memory space allocated for GPI

    \return an int where -1 is operation failed, -42 is timeout and 0 is success.

    \warning The command line arguments (argc, argv) won't be forward to the worker nodes

    */

    int startGPI(int argc,char *argv[],const char *cmdline,const unsigned long gpiMemSize)
```

After a successful GPI start, the application may query the current values on the node where it is running.

The function *getDmaMemPtrGPI* returns the address of the *GPI memory* block on the calling node. This address is guaranteed to be page-size aligned.

```
void *getDmaMemPtrGPI(void)
```

The binary executed is usually the same for all nodes. To distinguish between nodes in the source code a rank number is associated with every node. The rank of the master node is always zero, while the worker nodes are assigned integral numbers from one to the number of participating nodes minus one. After the GPI has been successfully started the rank of a node can be queried with *getRankGPI* whereas the number of nodes is given by *getNodeCountGPI*.

```
int getRankGPI(void)

int getNodeCountGPI(void)
```

At the end of an GPI application all resources associated with the GPI need to be released with a call to *shutdownGPI*.

```
void shutdownGPI()
```

The following commented code example shows a simple start and stop of GPI:

```
#include <GPI.h>
#include <GpiLogger.h>

#define GB 1073741824

int main(int argc, char *argv[])
{

  // start GPI with 1 GB memory
  if(startGPI(argc, argv, "", GB) != 0)
    {
      gpi_printf("GPI start-up failed\n");
      killProcsGPI();
      shutdownGPI();
```

```
      return -1;
    }

  //get rank
  const int rank = getRankGPI();

  //get number of nodes
  const int numNodes = getNodeCountGPI();

  //get pointer to global memory
  char * memPtr = (char *) getDmaMemPtrGPI();

  //everything up and running, syncronize
  barrierGPI();

  //shutdown
  shutdownGPI();

  return 0;
}
```

4.2 DMA operations

There are one-sided and two-sided DMA operations. The one-sided operations are *readDmaGPI* and *writeDmaGPI*. They are both non-blocking and do not require any involvement of the node read from or written to. The status of such an operation can only be checked by querying the associated queue on the calling node. The two-sided operations are *rcsvDmaGPI* and *sendDmaGPI*. For every *sendDmaGPI* there has to be a matching *recvDmaGPI* and vice versa. While the *sendDmaGPI* is also a non-blocking operation, *rcsvDmaGPI* will return only when the all the data has been transferred. This operation is useful where relaxed synchronisation is required between the sender and the receiver. As noted previously a DMA transfer is only possible to and from *GPI memory*. The source and destination memory locations of a DMA operation are not specified by pointers but by relative byte offsets from the *GPI memory* start addresses of the involved nodes. This makes DMA operations easy because the exact memory addresses are not required which would be different for all nodes.

```
int readDmaGPI(const unsigned long localOffset,const unsigned long remOffset,
               const int size,const unsigned int rank,const unsigned int gpi_queue)

int writeDmaGPI(const unsigned long localOffset,const unsigned long remOffset,
               const int size,const unsigned int rank,const unsigned int gpi_queue)

int sendDmaGPI(const unsigned long localOffset,const int size,

               const unsigned int rank,const unsigned int gpi_queue)
int recvDmaGPI(const unsigned long localOffset,const int size,
               const unsigned int rank,const unsigned int gpi_queue)
```

The arguments of all operations have identical arguments (where applicable):

localOffset The local offset where the data is transferred to/from.

remOffset The remote offset where the data is transferred to/from.

size The transfer size in bytes.

rank The node's rank where the data is transferred to/from.

gpiqueue The queue number to be used for the operation.

return An int where 0 is success and -1 is operation failed.

4.3 Queues

Every DMA operation requires a queue to be specified (either explicitly or implicitly). Every node has its own set of queues. Queues are used to organize and monitor DMA operations. Multiple DMA requests can be issued to the same queue and will be executed asynchrounously. The number of outstanding DMA operations in a queue can be determined with the function *openDMARequestsGPI*.

```
int openDMARequestsGPI(const unsigned int gpi_queue)
```

where

gpiqueue The queue number to check.

return An int with the number of open requests or -1 on error.

With *waitDmaGPI* it is possible to wait for all DMA operations of a queue to be finished.

```
int waitDmaGPI(const unsigned int gpi_queue)
```

gpiqueue The queue number to wait on.

return An int with the number of completed queue events or -1 on error.

As refered, each node has a given number of queues. This number of available queues is given by *getNumberOfQueuesGPI*.

```
int getNumberOfQueuesGPI(void)
```

Each queue allows a maximum number of outstanding DMA operations which is returned by *getQueueDepthGPI*.

```
int getQueueDepthGPI(void)
```

If this maximum number is reached every consecutive DMA request will generate an error. In such a case the queue is broken and cannot be restored. Always keep track of the number of requests posted to a queue or check its status with *openDMARequestsGPI* before executing a DMA operation. If a saturated queue is detected you have the following options: Call *waitDmaGPI* on the queue to wait for all operations to be finished, do some other work and try the same queue again later or use another empty queue. To see how multiple queues can be used to implement a buffered data transfer approach to overlap communication with computation have a look at the *bufferedtransfer* on Appendix B.

4.4 Passive DMA operations

The *sendDmaGPI* and *recvDmaGPI* operation also come in another flavour called passive DMA operations, namely *sendDmaPassiveGPI* and *recvDmaPassiveGPI*. The essential difference is that *recvDmaPassiveGPI* does not require the specification of a rank. Instead the operation waits for an incoming *sendDmaPassiveGPI* from *any* node. Once a connection has been established the sender can be identified with the *senderRank* argument.

```
int sendDmaPassiveGPI(const unsigned long localOffset, const int size, const unsigned int ↩
    rank)

int recvDmaPassiveGPI(const unsigned long localOffset, const int size, int *senderRank);
```

The arguments are similar to the other DMA operations:

localOffset The local offset where the data is transferred to/from.

size The transfer size in bytes.

rank The node's rank where the data is transferred to.

senderRank The rank of the node that sent the data or -1 if the sender could not be established.

return An int where 0 is success and -1 is operation failed.

The passive communication is useful when the communication pattern of an application is not known in advance. All passive DMA operations implicitly use a special passive queue. Monitoring this queue is possible with

```
int waitDmaPassiveGPI()

int openDMAPassiveRequestsGPI()
```

with the same semantics as for the queues of DMA queues, returning the number of completed events (*waitDmaPassiveGPI()*) and the number of open requests (*openDMAPassiveRequestsGPI()*).

4.5 Collective operations

GPI focuses on an asynchrounous programming model, trying to avoid collective and synchrounous operations at all. But some operations such as the allReduce are useful and make development easier.

At the moment, GPI only provides the allReduce collective operation. Contrary to the other communication calls, the application may give local buffers as input and output to the function (see below) instead of global offsets. The number of elements is limited to 255 (elemCnt) and the allowed operations and types are described below.

```
enum GPI_OP { GPI_MIN = 0, GPI_MAX = 1, GPI_SUM = 2 };

enum GPI_TYPE {GPI_INT = 0, GPI_UINT = 1, GPI_FLOAT = 2, GPI_DOUBLE = 3,
  GPI_LONG = 4, GPI_ULONG = 5};

int allReduceGPI(void *sendBuf, void *recvBuf, const unsigned char elemCnt, GPI_OP op, ↩
    GPI_TYPE type);
```

4.6 Synchronisation

GPI provides a fast barrier for a global synchronisation across all nodes.

```
void barrierGPI(void)
```

Another synchronization primitive is the global resource lock. All nodes can use it to limit access to a shared resource using the lock-unlock semantics. Once a node got the lock it can be sure it is the only one. Since it is a global resource is should be used wisely and in a relaxed manner (try not to busy-loop to get the lock).

```
int globalResourceLockGPI(void)
```

return An int where 0 is success (got lock) and -1 is operation failed (did not get lock).

```
int globalResourceUnlockGPI(void)
```

return An int where 0 is success and -1 is operation failed (not owner of the lock).

4.7 Atomic operations

GPI provides a limited number of atomic counters which are globally accessible from all nodes. The number of atomic counters available is returned by *getNumberOfCountersGPI*. Three atomic operations exist that can be used on the counters. The *atomicFetchAddCntGPI* operation will atomically add the *val* argument to the current value of the counter. The old value is returned. The *atomicCmpSwapCntGPI* operation will atomically compare the counter value with the argument *cmpVal* and in case they are equal the counter value will be replaced with the *swapVal* argument. The *atomicResetCntGPI* operation will simply set the counter value to zero. A special counter is the tile counter which has its own set of atomic functions that are technically identical to the standard atomic operations. The difference is just conceptual, you can use it as any other atomic counter.

```
int getNumberOfCountersGPI(void)

unsigned long atomicFetchAddCntGPI(const unsigned long val, const unsigned int gpi_counter)

unsigned long atomicCmpSwapCntGPI(const unsigned long cmpVal, const unsigned long swapVal, ↩
    const unsigned int gpi_counter)

int atomicResetCntGPI(const unsigned int gpi_counter)
```

4.8 Commands

Commands are simple 32-bit messages that can be sent between nodes. Command operations are always two-sided and blocking. To every sender corresponds one (or more) receiver(s). The function *setCommandGPI* can be called exclusively from the master node (rank 0) and will return only after all worker nodes have executed a matching *getCommandGPI*. Likewise a *getCommandGPI* will return only after a message from the master node has been received. Messages between any two nodes can be exchanged with *getCommandFromNodeIdGPI* and *setCommandToNodeIdGPI*. Since both function calls are blocking operations this mechanism can be utilized to synchronise between two nodes instead of using *barrierGPI* for a global synchronisation.

```
int getCommandGPI(void)

int setCommandGPI(const int cmd)

long getCommandFromNodeIdGPI(const unsigned int rank)

long setCommandToNodeIdGPI(const unsigned int rank, const long cmd)
```

4.9 Environment checks

The GPI offers a comprehensive set of environment checking functions that make it easy to detect problems with a GPI installation and help to separate programming errors from environment issues. These functions should be used *before* a call to *startGPI* is made. Hence it makes only sense to use them on the master node. The correct procedure is to identify the master node with *isMasterProcGPI* and then query the number of nodes with *generateHostlistGPI*. For each rank number from zero to the number of nodes minus one translate the rank to a hostname with *getHostnameGPI* and perform the various environment checks. Take a look at the *envtest* example on Appendix A. First check if the daemon of the node is reachable with *pingDaemonGPI*. Then verify if the port is free that is used by the daemons to communicate between nodes with *checkPortGPI* and *getPortGPI*. If this is not the case you can test if another GPI application is blocking the port with *findProcGPI*. Now test

with *checkSharedLibsGPI* if all required shared libraries are available on the node. The last step is to perform a basic network runtime check with *runIBTestGPI*.

```
int pingDaemonGPI(const char *hostname)

int isMasterProcGPI(int argc, char *argv[])

int checkSharedLibsGPI(const char *hostname)

int checkPortGPI(const char *hostname, const unsigned short portNr)

int findProcGPI(const char *hostname)

int clearFileCacheGPI(const char *hostname)

int runIBTestGPI(const char *hostname)
```

4.10 Configuring GPI

Before a call to *startGPI* has been made four important GPI parameters can be changed. With *setNetworkGPI* the network type is set to Infiniband or Ethernet. The default is Infiniband. The function *setPortGPI* allows to change the port used by the GPI for internal communication with the daemons. It is useful if another application is already using the default port. The MTU size for DMA transfers can be changed with *setMtuSizeGPI*. The default value is 1024 but you should use 2048 and above on modern cards. This brings a performance boost for data transfer. The function *setNpGPI* is useful if less nodes than those that are listed in the *machinefile* should run the GPI application.

```
int setNetworkGPI(GPI_NETWORK_TYPE typ)

int setPortGPI(const unsigned short port)

int setMtuSizeGPI(const unsigned int mtu)

void setNpGPI(const unsigned int np)
```

4.11 Notes on multi-threaded applications

Except for *recvDmaPassiveGPI* all GPI operations are thread-safe. It is advised that only a single thread has the function to do passive receives. Also care has to be taken to interpret the return values for *waitDmaGPI* and *waitDmaPassiveGPI* correctly. If the return value is zero it does not necessarily confirm that all DMA operations in a queue have been executed. Instead another thread may already be executing a *waitDma** on this queue.

A Code example - envtest.cpp

```
#include <GPI.h>
#include <GpiLogger.h>
#include <signal.h>
#include <assert.h>

#define GB 1073741824

void signalHandlerMaster(int sig)
{

  //do master node signal handling...
  //kill the gpi processes on all worker nodes, only callable from master
  killProcsGPI();
  //shutdown nicely
  shutdownGPI();

  exit(-1);
}

void signalHandlerWorker(int sig)
{

  //do worker node signal handling...
  //shutdown nicely
  shutdownGPI();

  exit(-1);
}

int checkEnv(int argc, char* argv[])
{

  int errors = 0;

  if(isMasterProcGPI(argc, argv) == 1){

    const int nodes = generateHostlistGPI();
    const unsigned short port = getPortGPI();

    //check setup of all nodes
    for(int rank=0; rank<nodes; rank++){

      int retval;

      //translate rank to hostname
      const char *host = getHostnameGPI(rank);

      //check daemon on host
      if(pingDaemonGPI(host) != 0){

        gpi_printf("Daemon ping failed on host %s with rank %d\n", host, rank);
        errors++;
        continue;
      }

      //check port on host
      if((retval=checkPortGPI(host, port)) != 0){

        gpi_printf("Port check failed (return value %d) on host %s with rank %d\n", retval, ↩
            host, rank);
        errors++;

        //check for running binaries
        if(findProcGPI(host) == 0){

          gpi_printf("Another GPI binary is running and blocking the port\n");
          if(killProcsGPI() == 0){

            gpi_printf("Successfully killed old GPI binary\n");
            errors--;
```

```
          }
        }
      }

      //check shared lib setup on host
      if((retval=checkSharedLibsGPI(host)) != 0){

        gpi_printf("Shared libs check failed (return value %d) on host %s with rank %d\n", ↩
            retval, host, rank);
        errors++;
      }

      //final test
      if((retval=runIBTestGPI(host)) != 0){

        gpi_printf("IB test failed (return value %d) on host %s with rank %d\n", retval, ↩
            host, rank);
        errors++;
      }
    }
  }

  return errors;
}


int main(int argc, char *argv[])
{

  //check the runtime eviroment
  if(checkEnv(argc, argv) != 0)
    return -1;

  //everything good to go, start the GPI
  if(startGPI(argc, argv, "", GB) != 0){

    gpi_printf("GPI start-up failed\n");
    killProcsGPI();
    shutdownGPI();
    return -1;
  }

  //get rank
  const int rank = getRankGPI();

  //setup signal handling
  if(rank != 0)
    signal(SIGINT, signalHandlerWorker);
  else
    signal(SIGINT, signalHandlerMaster);

  //print arguments, use the gpi_logger to view output on worker nodes
  for(int i=0; i<argc; i++)
    gpi_printf("argc: %d, argv: %s\n", i, argv[i]);

  //everything up and running, syncronize
  barrierGPI();

  //shutdown
  shutdownGPI();

  return 0;
}
```

B Code example - transferbuffer.cpp

```
#include <GPI.h>
#include <GpiLogger.h>
#include <MCTP1.h>
```

```
#include <signal.h>
#include <assert.h>
#include <cstring>

#define GB 1073741824
#define PACKETSIZE (1<<26)


void signalHandlerMaster(int sig){

  //do master node signal handling...
  //kill the gpi processes on all worker nodes, only callable from master
  killProcsGPI();
  //shutdown nicely
  shutdownGPI();

  exit(-1);
}


void signalHandlerWorker(int sig){

  //do worker node signal handling...
  //shutdown nicely
  shutdownGPI();

  exit(-1);
}


int checkEnv(int argc, char* argv[]){

  int errors = 0;

  if(isMasterProcGPI(argc, argv) == 1){

    const int nodes = generateHostlistGPI();
    const unsigned short port = getPortGPI();

    //check setup of all nodes
    for(int rank=0; rank<nodes; rank++){

      int retval;

      //translate rank to hostname
      const char *host = getHostnameGPI(rank);

      //check daemon on host
      if(pingDaemonGPI(host) != 0){

        gpi_printf("Daemon ping failed on host %s with rank %d\n", host, rank);
        errors++;
        continue;
      }

      //check port on host
      if((retval=checkPortGPI(host, port)) != 0){

        gpi_printf("Port check failed (return value %d) on host %s with rank %d\n", retval, ↩
            host, rank);
        errors++;

        //check for running binaries
        if(findProcGPI(host) == 0){

          gpi_printf("Another GPI binary is running and blocking the port\n");
          if(killProcsGPI() == 0){

            gpi_printf("Successfully killed old GPI binary\n");
            errors--;
          }
        }
      }
```

```
      //check shared lib setup on host
      if((retval=checkSharedLibsGPI(host)) != 0){

        gpi_printf("Shared libs check failed (return value %d) on host %s with rank %d\n", ↩
            retval, host, rank);
        errors++;
      }

      //final test
      if((retval=runIBTestGPI(host)) != 0){

        gpi_printf("IB test failed (return value %d) on host %s with rank %d\n", retval, ↩
            host, rank);
        errors++;
      }
    }
  }

  return errors;
}


int check(void *memptr, const unsigned long size){

  const char *ptr = static_cast<const char*>(memptr);

  for(unsigned long i=0; i<size; i++)
    if(ptr[i] != 1)
      return -1;

  return 0;
}


void doComputation(char *ptr, const unsigned long size){

  for(unsigned long j=0; j<size; j++)
      ptr[j]++;
}


int bufferedtransfer(void *memptr, const unsigned long packetsize, const int rank, const int↩
     nodecount){

  //permutation for send/work/recieve buffer
  const int permutation[] = {0,1,2,0,1};

  //buffers are located behind the data block in memory
  const unsigned long datasize = static_cast<unsigned long>(nodecount)*packetsize;
  const unsigned long bufferOffset[] = {datasize, datasize+packetsize, datasize+2*packetsize↩
      };
  const unsigned long workOffset = static_cast<unsigned long>(rank)*packetsize;
  //work buffer index
  int wIdx = 0;
  //store the node's rank of the data associated with a buffer
  int noderank[] = {0,0,0};

  //check for gpi errors
  int error = 0;

  //preload work buffer
  const int neighbour = (rank+1)%nodecount;
  error += readDmaGPI(bufferOffset[wIdx], workOffset, packetsize, neighbour, wIdx);
  noderank[wIdx] = neighbour;

  gpi_printf("preload: %i (node %i)\n", wIdx, neighbour);

  //do computation on local data
  char *ptr = static_cast<char*>(memptr) + workOffset;
  doComputation(ptr, packetsize);

  //work with remote data
  for(int i=2; i<(nodecount+1); i++){
```

```
    //the last round doesn't need preloading
    if(i < nodecount){

      const int nr = (rank+i)%nodecount;
      const int bIdx = permutation[wIdx+1];

      //preload the second next work buffer
      if(waitDmaGPI(bIdx) == -1 )
        error += -1;

      error += readDmaGPI(bufferOffset[bIdx], workOffset, packetsize, nr, bIdx);
      noderank[bIdx] = nr;

      gpi_printf("preload: %i (node %i)\n", bIdx, nr);
    }

    //wait for the work buffer to finish receiving
    if(waitDmaGPI(wIdx) == -1)
      error += -1;

    //do computation
    char *ptr = static_cast<char*>(memptr) + bufferOffset[wIdx];
    doComputation(ptr, packetsize);

    //send back
    error += writeDmaGPI(bufferOffset[wIdx], workOffset, packetsize, noderank[wIdx], wIdx);

    gpi_printf("send: %i (node %i)\n", wIdx, noderank[wIdx]);

    //switch to next work buffer
    wIdx = permutation[wIdx+1];
  }

  //wait for all queues to finish
  if(waitDmaGPI(permutation[wIdx+1]) == -1)
    error += -1;

  if(waitDmaGPI(permutation[wIdx+2]) == -1)
    error += -1;

  return error;
}


int main(int argc, char *argv[]){

  //check the runtime eviroment
  if(checkEnv(argc, argv) != 0)
    return -1;

  //mctp timer for high resolution timing
  mctpInitTimer();

  //everything good to go, start the GPI
  if(startGPI(argc, argv, "", GB) != 0){

    gpi_printf("GPI start-up failed\n");
    killProcsGPI();
    shutdownGPI();
    return -1;
  }

  //get rank
  const int rank = getRankGPI();

  //setup signal handling
  if(rank != 0)
    signal(SIGINT, signalHandlerWorker);
  else
    signal(SIGINT, signalHandlerMaster);

  //init memory
  const int nodecount = getNodeCountGPI();
```

```
  void *memptr = getDmaMemPtrGPI();
  memset(memptr, 0, nodecount * PACKETSIZE);

  //everything up and running, syncronize
  barrierGPI();

  mctpStartTimer();

  if(bufferedtransfer(memptr, PACKETSIZE, rank, nodecount) != 0)
    gpi_printf("Communication error\n");

  //everything finished, syncronize
  barrierGPI();

  mctpStopTimer();

  const unsigned long tsize = 2 * static_cast<unsigned long>( nodecount - 1 ) * PACKETSIZE;
  gpi_printf("Transfered %u bytes between %u nodes in %f msecs (%f GB/s)\n",
         tsize,
         nodecount,
         mctpGetTimerMSecs(),
         tsize/1073741824.0/mctpGetTimerSecs());

  //check for errors
  if(check(memptr, nodecount * PACKETSIZE) != 0)
    gpi_printf("Check failed\n");

  //shutdown
  shutdownGPI();

  return 0;
}
```